Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Benjamin Muhič

Avtomatizacija funkcionalnega testiranja v procesu razvoja programske opreme

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Luka Šajn

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

[1]V dogovorju z mentorjem lahko kandidat diplomsko delo s pripadajočo izvorno kodo izda tudi pod katero izmed alternativnih licenc, ki ponuja določen del pravic vsem: npr. Creative Commons, GNU GPL. V tem primeru na to mesto vstavite opis licence, na primer tekst [1].

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite področje razvoja in testiranja programske opreme s poudarkom na avtomatskem funkcionalnem testiranju. Navedite razloge za vpeljavo avtomatskega testiranja ter opišite življenjski cikel le-tega po metodologiji ATLM. Preučite eno izmed možnosti implementacije avtomatskega funkcionalnega testiranja v podjetju ter opišite uporabljeno orodje za avtomatizacijo preko grafičnega uporabniškega vmesnika.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Benjamin Muhič, z vpisno številko **63110301**, sem avtor diplomskega dela z naslovom:

Avtomatizacija funkcionalnega testiranja v procesu razvoja programske opreme
Automation of functional testing in software development process

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Luke Šajna,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 28. avgusta 2014 Podpis avtorja:

Hvala moji družini, da me je na študijski poti moralno in finančno podpirala ter mi vseskozi stala ob strani.

Hvala mojemu dekletu Špeli Zajec, za vso ljubezen, podporo in spodbudo tekom študijskih let.

Hvala tudi prijateljem: Albinu, Juretu, Roku, Gregi, Davorju in Simonu za nesebično pomoč, veliko neprespanih študijskih noči in lepih trenutkov.

Hvala podjetju Halcom d.d. za vso izkazano podporo, spodbudo, razumevanje in nakup orodja za avtomatizacijo.

Najlepša hvala mentorju doc.dr. Luki Šajnu, ki me je med izdelavo naloge vseskozi usmerjal ter mi pomagal z nasveti in pripombami.

Hvala tudi vsem ostalim, ki so kakorkoli prispevali k nastanku tega diplomskega dela.

Kazalo

Slike

Povzetek

V današnjem hitro razvijajočem svetu in spremenjenim zahtevam na trgu, se vse več podjetij odloča za agilne metode razvoja programske opreme. Glavni namen uporabe agilnih metod je predvsem hitro prilaganje razvojnega procesa in spremembam na trgu. Zaradi kratkih časovnih rokov za dobavo produktov, si podjetja želijo skrajšati cikel testiranja programske opreme, kljub vsemu pa še vedno ustvarjati in dobavljati kvalitetne produkte. Zato se odločajo za avtomatizacijo testiranja programske opreme na različnih ravneh, ki ob smotrni vpeljavi prinaša veliko koristi.

Cilj diplomske naloge je preučiti, katero orodje je najbolj primerno za avtomatizacijo funkcionalnega testiranja v podjetju Halcom, kakšne so prednosti tega orodja, kdaj je smiselno začeti s testiranjem, kakšen je proces testiranja in ugotavljanje koristi vpeljave avtomatskih testov. Ključne besede: ATLM, avtomatizacija testiranja, Ranorex, metode testiranja, funkcionalno testiranje, testiranje modulov.

Ključne besede: ATLM, avtomatizacija testiranja, Ranorex, metode testiranja, funkcionalno testiranje, testiranje modulov.

Abstract

Rapid development and changes in technology and increasing market demands are leading companies to choose agile methods in software development. The main objective of applying agile methods is primarily fast adaptation of development process to changed market demands. Short deadlines in the product supply chain, demand from the companies on one side, to shorten the software testing cycle, but on the other side to develop and produce high quality products. Therefore companies decide on automating software testing at various levels, which can result in many benefits when introduced efficiently. The aim of the thesis is to examine which tool is most suitable for automation of functional testing in company Halcom. Thesis depicts, tool's advantages, when it makes sense to introduce automation testing, the testing process and identifies the benefits of introducing automated tests.

Keywords: ATLM, software testing automation, Ranorex, testing methods, functional testing, unit testing.

Poglavje 1

Uvod

Vse večja kompleksnost sistemov, stroškovni ter časovni okviri, predstavljajo izziv testiranju programske opreme. Ena izmed ključnih težav je, da testiranje ponavadi nastopi v izdihljajih življenskega cikla projekta, ter da se izvaja ročno. Kadar se pri testiranju programske opreme pojavijo napake je slednje potrebno odpraviti in ponoviti testiranje, zato se stroški projekta naglo povečujejo.

Zaradi potrebe po večji produktivnosti in zmanjševanju stroškov se vse večkrat srečamo z agilnim razvojem programske opreme. Agilni razvoj omogoča hitro prilagajanje spremembam zahtev naročnika. Hitro prilagajanje spremembam je mogoče zaradi kratkih in pogostih iteracij, tesnega sodelovanja z naročnikom in načrtovanja z malo dokumentacije. Danes najbolj poznani metodi agilnega razvoja programske opreme sta **Scrum** ter hibrid med metodo Scrum in metodo **Extremnega programiranja** (angl. *Extreme Programming*).

Kljub vse večjemu posluževanju agilnega razvoja, avtomatizacija testiranja ni odvisna od metodologije razvoja programske opreme. Z ustrezno implementacijo avtomatskega testiranja je slednje koristno ne glede na izbrano metodologijo razvoja programske opreme, še posebno kadar gre za regresijsko testiranje, katero skrbi za zagotavljanje kakovosti in stabilnosti programske opreme po spremembah.

Poznamo veliko različnih pristopov k avtomatskemu testiranju, v pričujočem delu pa nameravam preučiti avtomatsko funkcionalno testiranje preko grafičnega uporabniškega vmesnika (angl. *Graphical User Interface*). Avtomatizacija testiranja preko grafičnega uporabniškega vmesnika preverja delovanje vseh funkcij ter akcij v aplikaciji, ki jih končni uporabnik lahko izvede, največkrat pa se uporablja pri aplikacijah, ki se drastično ne spreminjajo oz. se jim kasneje v življenskem ciklu doda zgolj kakšen manjši popravek ali dopolnitev, saj nam to zagotavlja regresijo testiranja.

1.1 Motivacija za izdelavo diplomske naloge

V podjetju Halcom smo se po številnih analizah ter želji po izboljšavah v procesu testiranja, odločili za vpeljavo avtomatizacije testiranja, kar je tudi spodbudilo pisanje tega diplomskega dela. Z avtomatizacijo testiranja smo želeli predvsem skrajšati življenski cikel testiranja, hitrejšo odzivnost testiranja po dnevnih izdajah programske kode, regresijsko testiranje, lažje vzdrževanje, ter izvajanje testov na različnih operacijskih sistemih in platformah.

Poglavje 2

Razvoj programske opreme

2.1 Življenski cikel razvoja programske opreme

Kot večina razvojnih procesov ima tudi razvoj programske opreme svoj življenjski cikel(angl. Software Development Life Cycle), ki predpisuje zaporedje faz razvoja. V vsaki izmed faz razvoja se odvijajo različne aktivnosti:

- **Analiza zahtev** je namenjena predvsem popolnemu razumevanju nalog in lastnosti programske opreme. Vse zahteve je potrebno skrbno dokumentirati in se o njih pogovoriti z naročnikom.

- **Načrtovanje** programske opreme je namenjeno takšni predstavitvi programske opreme, da njene lastnosti lahko ocenimo še preden se kodiranje začne.

- **Kodiranje** predstavlja implementacijo načrtovanih zahtev.

- **Testiranje** se mora izvajati ves čas razvoja programske opreme, saj prej ko ugotovimo napako ali pomanjkljivost med razvojem, hitreje in z manj stroški jo lahko odpravimo. *Verifikacija* je preverjanje posamezne faze v razvoju. *Validacija* pa vzame v obzir širši vidik in preverja, ali so rezultati posamezne faze skladni z osnovnimi zahtevami, postavljenimi na začetku razvoja.

- **Vzdrževanje** programske opreme je potrebno zaradi odpravljanja napak po končanem razvoju in zaradi spremenjenih zunanjih okoliščin, v katerih sistem deluje.

2.2 Modeli razvoja programske opreme

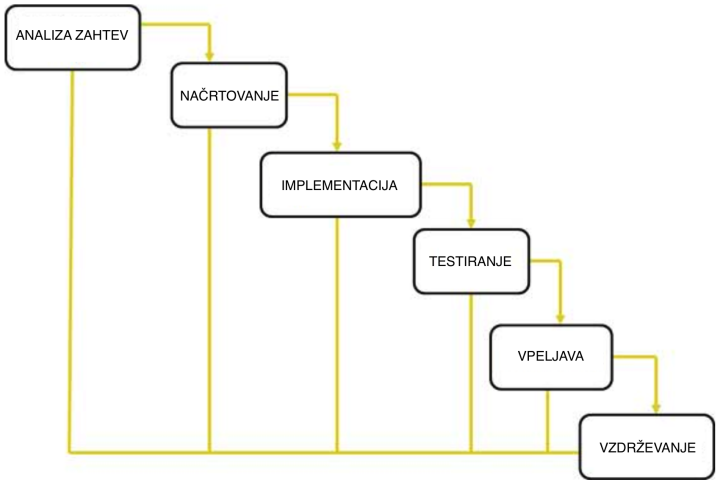
Poznamo različne življenske modele razvoja programske opreme (slapovni model, spiralni model, V-model, agilne metodologije,...), v praksi pa se večinoma uporablja kombinacija različnih modelov, med katerimi trenutno prevladujejo modeli agilnega razvoja.

2.2.1 Zaporedni ali slapovni model

Zaporedni ali slapovni model je eden izmed prvih življenskih modelov, ki je bil uporabljen za uspešen razvoj programske opreme. Je zelo enostaven za razumevanje, saj si faze sledijo zaporedno. Naslednja faza se začne, ko se predhodna konča. Faze se med seboj ne prekrivajo.

Njegova ključna prednost je, da natanko definira aktivnosti posameznih faz razvoja in izdelke, ki so potrebni za prehod v naslednjo fazo. S tem definira hrbtenico za razvoj kompleksnih aplikacij. Kljub dejstvu, da zaporedni model uvaja disciplinirano izvajanje aktivnosti posameznih faz, ki so dobro dokumentirane, testirane in pregledane, se danes v taki obliki ne uporablja več.

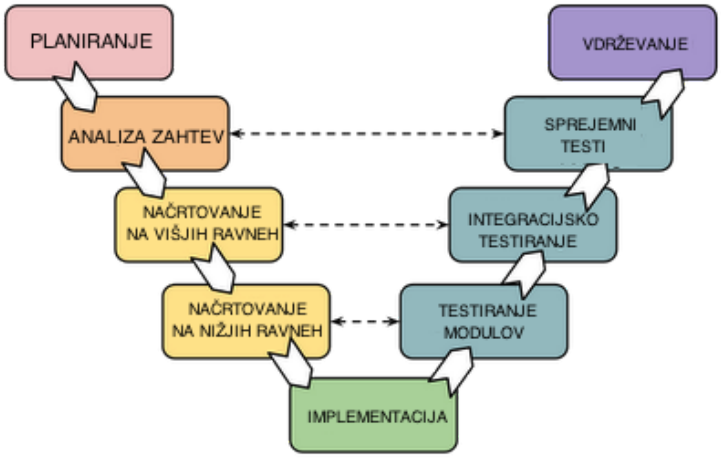
Njegova glavna pomanjkljivost je, da ne odraža resničnega razvojnega procesa, saj model kot tak ne omogoča vračanja v predhodne faze, kar pa se v praksi pogosto dogaja.



Slika 2.1: ***Slapovni model*** razvoja programske opreme.

2.2.2 V-model

V-model je nadgradnja slapovnega modela, katerega cilj je odkriti napake kar se da zgodaj. To zahteva skrbno analizo in sodelovanje naročnika ali uporabnika. Da ne bi prišlo do prevelikega odstopanja med sistemom, ki ga razvijamo, in zahtevami uporabnikov, je smiselno po vsaki fazi poleg verifikacije rezultatov izvesti tudi njihovo validacijo.

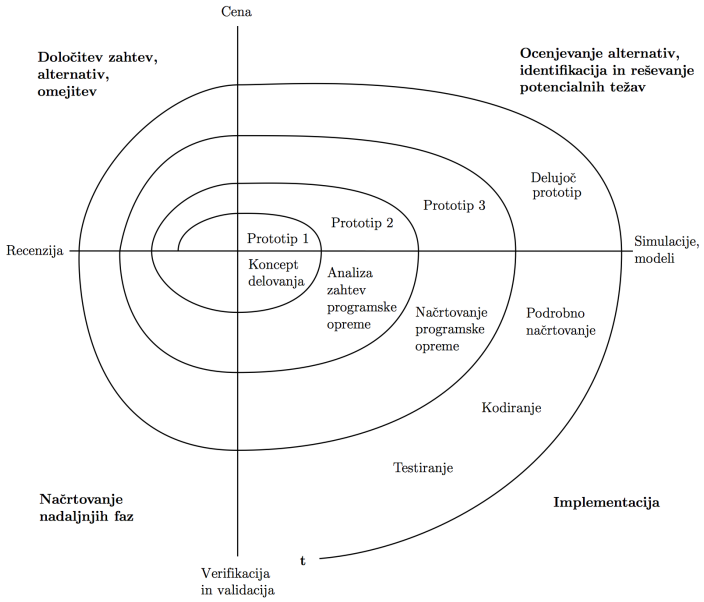


Slika 2.2: ***V-model*** razvoja programske opreme.

2.2.3 Spiralni model

Spiralni model razvoja, kot že samo ime pove, vsebuje več ciklov, od katerih vsak vsebuje fazo analize, načrtovanja, implementacije in testiranja. Prvi od teh ciklov so namenjeni razvoju prototipov, kasnejši pa za adaptacijo obstoječih sistemov. Bohem, ki je predlagal spiralni model razvoja programske opreme, je vanj vključil vse do tedaj opisane modele razvoja[20]:

- Vsaka napaka ali zahteva sproži nov obhod spirale.
- Če so zahteve jasne in želimo zgraditi robusten in dobro dokumentiran sistem, sledimo spirali enkrat in uporabimo klasični razvojni cikel.
- Postopno lahko zgradimo sistem tako, da spiralo večkrat obkrožimo, vsakokrat za en del sistema.
- Če so zahteve nejasne, lahko spirali sledimo tolikokrat, da s pomočjo prototipov rešimo problem.



Slika 2.3: ***Spiralni model*** razvoja programske opreme[20].

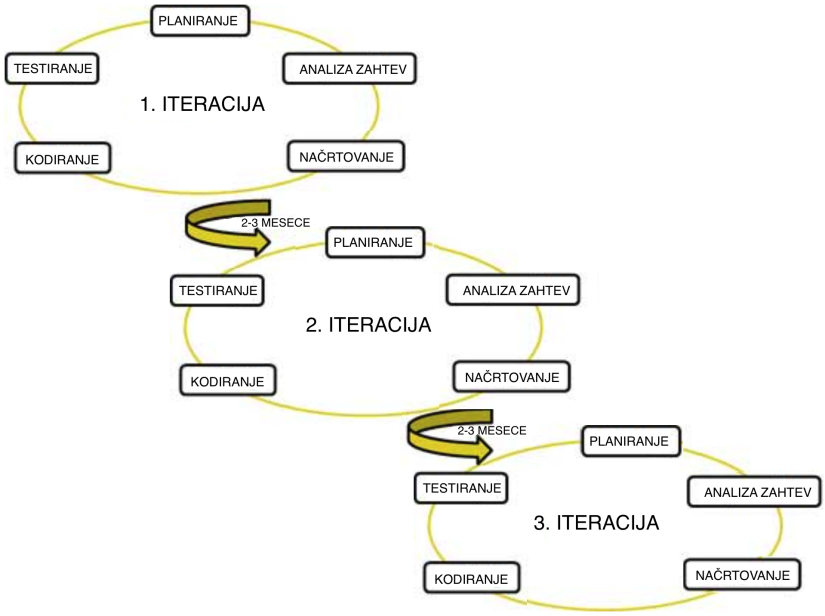
2.2.4 Agilne metodologije razvoja programske opreme

Dandanes se vse več podjetij pri razvoju projektov opira na tako imenovane agilne metodologije razvoja programske opreme. Trenutno najbolj uporabljeni metodi agilnega razvoja sta *Scrum* ter *Ekstremno programiranje*. Cilj agilnih metodologij je postavljanje prioritete na razvoj produkta, delujočo programsko kodo, prenos odgovornosti na razvojno skupino ter na zavedanje, da se uporabniške zahteve s časom spreminjajo [14].

Eno izmed načel ekstremnega programiranja je, da je potrebno teste za programsko opremo napisati, predno začnemo s kodiranjem. Ekstremno programiranje sledi logiki, da v kolikor nismo sposobni napisati testov, to pomeni, da ne poznamo natančnega delovanja izdelka, da o njem nimamo zadosti informacij, torej ni pravega smisla, da bi z razvojem sploh začeli. Strmi tudi k pogostemu poganjanju testov, še posebaj v primerih večjih sprememb v programski kodi, saj se s tem zaščitimo pred nevarnostjo, da bi poškodovali že delujoče dele kode [8].

Slika 2.4: *Glavne značilnosti metodologije* ***ekstremnega programiranja.***

Zaradi potreb po pogostejšem poganjanju testov, je smiselno razmisliti o avtomatskem testiranju programske kode, če ne celo nujno. Z avtomatskimi testi namreč dosežemo hitro odzivnost testiranja po dnevnih izdajah programske kode in s tem omogočimo razvojni skupini, da relativno hitro dobi



Slika 2.5: ***Agilni model*** razvoja programske opreme.

odziv, ali koda ustrezno deluje, kar z ročnim testiranjem praktično ni mogoče doseči v zglednem času. Poleg hitrejšega testiranja, se avtomatski testi lahko uporabijo večkrat, pri testiranju pa ni obvezna prisotnost testerja, zato se slednje lahko izvaja kadarkoli.

Poglavje 3

Testiranje programske opreme

S testiranjem programske opreme naročniku zagotovimo dobavo kakovostnega izdelka ali storive, ki jo razvijamo. Testiranje se izvaja s pomočjo predpisanih testnih primerov, s katerimi zagotovimo zanesljivo delovanje produkta, ki je največkrat namenjen končnim uporabnikom.

V tem poglavju bom opisal najbolj zanimive metode in postopke testiranja v zadnjih dvajsetih letih ter podal nekaj dejstev, zaradi katerih je vpeljava avtomatizacije testiranja v okoljih IT podjetji vse pogostejše.

3.1 Metode testiranja programske opreme

V grobem ločimo dva načina testiranja programske opreme:

- Statično testiranje
- Dinamično testiranje

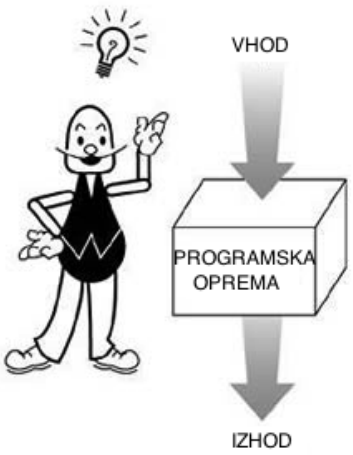
Pri **statičnem testiranju** gre za ročno pregledovanje dokumentacije ter kode, kar je največkrat dolgotrajen proces, ki se izvaja in ureja na koncu projekta.

V nasprotju s statičnim testiranjem **dinamično testiranje** poteka že med samim razvojem produkta, saj lahko z izvajanjem programa sproti preverjamo zaključene dele kode oziroma module.

Osnovna pristopa k testiranju programske opreme sta ***strukturno testiranje ali metoda bele skrinjice*** in ***funkcionalno testiranje ali metoda črne skrinjice***, *poznana pa je tudi metoda sive skrinjice, ki je mešanica obeh osnovih pristopov [15].*

3.1.1 Metoda bele skrinjice

Metoda bele skrinjice ali strukturno testiranje je verifikacijska tehnika, s katero programski inženirji preverjajo pričakovano delovanje napisane kode, in je hkrati osnova za pisanje testnih primerov.



Slika 3.1: Pri metodi *bele skrinjice* poznamo vsebino skrinjice.

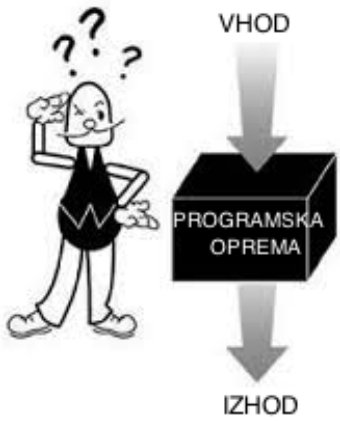
3.1.2 Metoda črne skrinjice

Pri **metodi črne skrinjice** ali funkcijskem testiranju pa testerje zanima samo funkcijska sposobnost programske opreme, torej preverjajo samo delovanje funkcionalnosti, ki jih mora programska oprema omogočati, ne ukvarjajo pa se s tem kako so funkcionalnosti razvite v programski kodi.

Funkcionalnosti, ki jih testirajo, so največkrat napisane v obliki funkcijskih specifikacij, ki so v naprej dogovorjene z naročnikom produkta. Te-

stiranje ponavadi poteka tako, da v aplikacijo vnesemo vhodne podatke ter primerjamo izhodne podatke z pričakovanimi. Funkcionalno testiranje ni nadomestilo za strukturirano testiranje, saj odkriva druge vrste napak, npr.:

- napake pri procesiranju
- manjkajoče funkcije
- napake vmesnika
- napake pri inicializaciji



Slika 3.2: Pri metodi *črne skrinjice* nas vsebina skrinjice ne zanima.

3.1.3 Metoda sive skrinjice

Testiranje po **metodi sive skrinjice** je kombinacija testiranja, po metodi bele in črne skrinjice. Pri tem načinu testiranja je testerju poznan del delovanja programa v ozadju, kar mu omogoča boljšo pripravo testnih primerov, četudi jih nato preverja po metodi črne skrinjice.

Oba osnovna pristopa sta torej enakovredna in se dopolnjujeta. Uporabljata se lahko že v relativno zgodnjih fazah razvoja programske opreme

in oba lahko avtomatiziramo, kar nam zagotavlja zgodnjo odpravo napak v produktu.

3.2 Tipi testiranja programske opreme

Ko govorimo o testiranju programske opreme, poznamo šest osnovnih tipov testiranja, ki pokrivajo veliko področje razvoja programske opreme. Vsak izmed tipov testiranja ima svoje specifične lastnosti, ki določajo kakšno je pravilno oz. napačno delovanje programske opreme.

3.2.1 Testiranje modulov

Testiranje modulov se izvaja po metodi bele skrinjice, in sicer največkrat v fazi verifikacije programske opreme. Izvaja se na posameznih delih strojne opreme oz. zaključenih celotah programske opreme, najboljše je da so moduli samostojni, torej neodvisni od ostalih modulov [22]. Testiranje modulov ponavadi testirajo razvijalci, npr. razvijalec napiše skupek testne kode, ki metodi v kodi pošje točno določene parametre in preveri, če je metoda vrnila ustrezne rezultate. Testiranje posameznih delov kode prinaša kar nekaj koristi, nekatere izmed njih so:

- **Odkrivanje napak v zgodnji fazi razvoja**
- **Regresijsko testiranje**
- **Preprostejša integracija**
- **Dokumentiranje**

3.2.2 Integracijsko testiranje

Integracijsko testiranje se izvaja tako na nizkem nivoju razvoja programske opreme, kot tudi na višjih nivojih. Za testiranje tega tipa se uporabljata metodi črne skrinjice in bele skrinjice. Pri integracijskem testiranju želimo

ovrednotiti kako dobro delujejo povezave med moduli programske ter strojne opreme, ko je le ta integrirana v večji skupek programske kode. Delovanje posameznih modulov, ki so izolirani od ostale programske opreme, ki je skupek nekega produkta, še ne pomeni, da bojo dobro delovali tudi po integraciji v skupen repozitorij. Velikokrat se namreč zgodi, da se podatki na poti izgubijo, bodisi zaradi napačno implementiranih vmesnikov ali kakšne druge napake pri integraciji programske opreme.

3.2.3 Funkcijsko in sistemsko testiranje

Pri funkcijskem in sistemskem testiranju se uporablja metoda črne skrinjice. Izvaja se na višjem nivoju razvoja programske opreme, za testiranje pa so potrebne specificirane zahteve delovanja produkta, ki so usklajene z naročnikom. Funkcijsko testiranje zagotavlja, da delujejo vse funkcionalnosti, ki so predpisane v funkcijskih specifikacijah. Sistemsko testiranje po drugi strani zajema namestitev programske opreme v različna okolja in s tem zagotavlja, da bo program pravilno deloval v naročnikovem okolju in na različnih verzijah operacijskih sistemov. Med funkcijsko in sistemsko testiranje uvrščamo:

- **Stresne teste.** S stresnimi testi preverjamo delovanje sistema na robu njegovih zmogljivosti. Pri testiranju se maksimalno obremeni sistem in se nato preverja njegovo robustnost, dosegljivost, odziv na napake,itd.

- **Testiranje odzivnosti (angl. Performance testing).** Pri testiranju odzivnosti pod drobnogled vzamemo moduli, kateri imajo specifično predpisano hitrost odzivanja.

- **Testiranje uporabnosti (angl. Usability testing).** Testiranje uporabnosti največkrat poteka v interakciji človek-računalnik, katerega izvaja tester.

3.2.4 Potrditveni test

Po funckijskem in sistemskem testiranju se produkt preda naročniku, kateri izvede testiranje produkta na svojem okolju, ki temelji na njegovem pričakovanju in funkcijskih specifikacijah. Naročniki ponavadi niso izkušeni testerji, tako da ne pregledajo celotnega obsega testnih primerov, ki predpisujejo ustrezno delovanje produkta. V primeru, da produkt ne prestane potrditvenega testa, pa si pridružujejo pravico zavrnitve produkta.

3.2.5 Regresijsko testiranje

Regresijski testi predstavljajo nekakšno podmnožico testov celotnega sistema, ki se izvajajo vsakokrat, ko pridejo nove dnevne oziroma tedenske izdaje produkta (angl. *Build*). Namen regresijskih testov je preverjanje pravilnega delovanja nove kode, ki je bila dodana v obstoječo rešitev, ter zagotavljanje, da obstoječa koda še vedno deluje pravilno.

3.2.6 Beta testiranje (angl. Beta testing)

Po končanem razvijanju produkta, se lahko razvojna skupina odloči, da ponudi določeno število svoje programske rešitve brezplačno. V zameno pričakujejo povratne informacije potencialnih uporabnikov ter prijavo napak, ki so jih uporabniki odkrili pri uporabi produkta. Pozitivni učinki beta testiranja so:

- *Odkrivanje nepričakovanih napak* saj produkt testira veliko število različnih uporabnikov
- *Testiranje na različni operacijskih sistemih*
- *Nizki stroški testiranja*

Poglavje 4

Avtomatsko testiranje

Čeprav se z ročnimi testi ugotovi veliko napak v programski opremi, pa je ta proces zahteven in dolgotrajen, poleg tega pa morda še neučinkovit pri iskanju določenih napak. Avtomatizacija testiranja je proces pisanja računalniškega programa, ki nadomesti testiranje programske opreme, ki bi sicer moralo biti izvedeno ročno. Ko so testi avtomatizirani se jih lahko izvaja relativno hitro, poleg tega pa so ponovljivi. Avtomatizacija testiranja je stroškovno najbolj učinkovita pri projektih, ki imajo daljšo življensko dobo, ter hkrati daljšo dobo vzdrževanja. S časom se namreč lahko pojavijo napake, ki jih v začetku življenskega cikla ni bilo, avtomatski testi pa nam omogočajo agilen odziv na napake [4].

Poznamo dva osnovna pristopa k avtomatskemu testiranju programske opreme, ***avtomatsko testiranje modulov***, ki poteka po metodi črne in bele skrinjice, testira pa delovanje posameznih modulov v programski kodi, ter ***avtomatsko funkcionalno testiranje***, ki simulira testiranje funkcionalnosti aplikacije preko grafičnega uporabniškega vmesnika. V pričujočem delu se bom nekoliko bolj osredotočil na avtomatsko funkcionalno testiranje.

4.1 Avtomatsko testiranje modulov

Ekstremno programiranje je kot metodologija agilnega razvoja programske opreme popolnoma spremenila vlogo testiranja modulov, saj ga v svojem življenskem ciklu uvršča med glavne faze razvoja programske opreme.

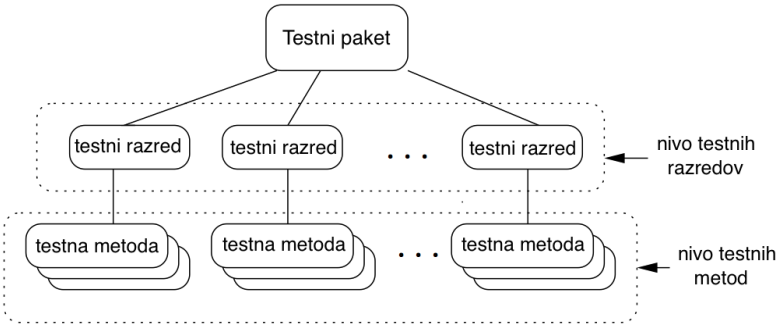
Testiranje modulov omogoča razvojnikom, da zaupajo delovanju napisane programske kode, saj morajo biti testi venomer v celoti uspešni. Izvedeni morajo biti ob vsaki novi integraciji kode v sistem, saj je potrebno v primeru napake že pri samo enem izmed modulov, slednjo odpraviti še predno nadaljujemo z integracijo. Testi se implementirajo vzporedno s pisanjem programske kode in se pozneje po potrebi spreminjajo ali dopolnjujejo. To je zelo pomembno pri metodologiji ekstremnega programiranja, saj neprestana integracija pomeni, da mora biti nekajkrat dnevno v sistem vključena vsa popravljena koda, kar je eden izmed glavnih razlogov za avtomatizacijo testiranja modulov.

Trend pri avtomatskem testiranju modulov je uporaba ogrodja (angl. *framework*) JUnit, ki je napisan v Javi. Omogoča testiranje modulov, s katerim ugotavlja, ali se različni deli kode pod različnimi pogoji obnašajo pričakovano.

4.1.1 Testiranje modulov z ogrodjem JUnit

JUnit je bilo prvo ogrodje, ki je razširilo uporabo testiranja modulov. Je ogrodje za avtomatsko testiranje delov kode, ki ga uporabljajo razvijalci, ter s tem povečujejo hitrost in kvaliteto napisane programske kode [13].

Testni primeri v JUnit ogrodju so Java razredi, ki vsebujejo eno ali več testnih metod, in so povezani v testne pakete [2]. Slika 4.1 prikazuje preprosto hierarhijo, ki ima zgolj en nivo testnih razredov, čeprav je nivojev testnih razredov lahko več, najmanjša enota v drevesu pa je testna metoda.



Slika 4.1: *Struktura testnega paketa JUnit ogrodja*

V nadaljevanju je predstavljen preprost primer implementacije JUnit testa v Javi s pozitivnim in negativnim scenarijem. Definirali bomo tri javanske razrede s katerimi bomo preverili, ali se vrednosti spremenljivke *sporočilo* ujemata.

Za začetek naredimo javanski razred, ki vsebuje konstruktor, ki nam bo priredil vrednost spremenljivke ter metodo za izpis spremenljivke.

```
public class MessageUtil {

   private String sporocilo;

   //Konstruktor
   public MessageUtil(String sporocilo){
      this.message = message;
   }

   //izpise sporocilo
   public String izpisiSporocilo(){
      System.out.println(sporocilo);
      return sporocilo;
   }
}
```

Za zgornjo kodo pripravimo testni primer, ki bo vseboval metodo za testiranje sporočila. Nad metodo, ki bo preverjala ali sta sporočili enaki, moramo dodati anotacijo **@Test**, ki označuje, da se javna metoda, katero označuje, lahko izvede kot testni primer.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestniRazred {

   String sporocilo = "Pozdravljen!";
   MessageUtil messageUtil = new MessageUtil(sporocilo);



   @Test
   public void testIzpisiSporocilo() {
      assertEquals(sporocilo,messageUtil.izpisiSporocilo());
   }
}
```

Preverjanje rezultatov izvajamo s pomočjo trditev (angl. *asserts*). Kot lahko opazimo se v našem testnem razredu znotraj metode *testIzpisiSporocilo()* pojavi trditev *assertEquals()*, ki bo preverila, če je rezultat enak predhodno določeni vrednosti. Trditve so vnaprej definirane metode, s katerimi preverjamo resničnost postavljenih pogojev. Po izvedbi testov so prikazane samo trditve pri katerih se je izkazalo, da rezultat ne ustreza postavljenim pogojem. Nekatere pomembnejše trditve so:

- **void assertEquals(boolean pričakovana, boolean dejanska)** - trditev je uspešna, če je rezultat enak predhodno določeni vrednosti
- **void assertTrue(boolean pričakovana, boolean dejanska)** - trditev je uspešna, če je pogoj resničen
- **void assertFalse(boolean pričakovana, boolean dejanska)** - trditev je uspešna, če je pogoj neresničen

- **void assertNull(Object avtomobil)** - trditev je uspešna, če ima argument nedefinirano vrednost

Definiramo še razred, ki nam omogoča izvedbo testov s pomočjo metode *runClasses(TestniRazred.class)*, ki je del razreda *JUnitCore*. Poleg tega pa uporabimo tudi objekt *Result* za obdelavo in izpis rezultatov.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class IzvediTest {
   public static void main(String[] args) {

      Result rezultat = JUnitCore.runClasses(TestniRazred.class);

      for (Failure napaka : rezultat.getFailures()) {
         System.out.println(napaka.toString());
      }

      System.out.println(rezultat.wasSuccessful());
   }
}
```

Ko izvedemo test, preverimo rezultat, ki v našem primeru vrne:

- **Pozdravljeni!**, ki je rezultat metode *izpisiSporocilo()*, ter
- **true**, ki je rezultat metode *System.out.println(rezultat.wasSuccessful());*

Rezultati kažejo, da se je test uspešno izvedel. Popravimo *TestniRazred* tako, da se testi ne bodo izvedli uspešno. V metodi *testIzpisiSporocilo* spremenimo vrednost spremenljivke *sporocilo*.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
```

```
public class TestniRazred {

   String sporocilo = "Pozdravljen!";
   MessageUtil messageUtil = new MessageUtil(sporocilo);


   @Test
   public void testIzpisiSporocilo() {
      //spremenimo vrednost spremenljivke sporocilo
      sporocilo = "Lep pozdrav!";
      assertEquals(sporocilo,messageUtil.izpisiSporocilo());
   }
}
```

Ostale razrede pustimo take kot smo jih sprogramirali na začetku in poženemo razred *IzvediTest*.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class IzvediTest {
   public static void main(String[] args) {

      Result rezultat = JUnitCore.runClasses(TestniRazred.class);

      for (Failure napaka : rezultat.getFailures()) {
         System.out.println(napaka.toString());
      }

      System.out.println(rezultat.wasSuccessful());
   }
}
```

Po tem, ko smo spremenili vrednost spremenljivke *sporocilo*, dobimo sledeč rezultat:

- **Pozdravljeni!**, ki je rezultat metode *izpisiSporocilo()*, razreda *MessageUtil*, katerega nismo spreminjali,
- **testPrintMessage(TestniRazred): expected:<[Lep pozdrav!]> but was:<[Pozdravljeni!]>**, ki je rezultat metode *napaka.toString()*, razreda *IzvediTest*, ter
- **false**, ki ga vrne metoda *rezultat.wasSuccessful()*

Kot vidimo nam JUnit omogoča relativno enostavno programiranje testnih metod, s katerimi preverjamo delovanje posameznih delov napisane programske kode. Omogoča nam tudi jasen prikaz rezultatov, med katerimi takoj vidimo na katerem delu kode je prišlo do napake, kar nam omogoča hitro odpravo napak ter lažjo nadaljno integracijo kode v sistem. Poleg vsega že naštetega podajmo še nekaj razlogov, zakaj bi se odločili za avtomatizacijo testiranja modulov z ogrodjem JUnit:

- **Programski jezik je enak testnemu**, saj je JUnit ogrodje napisano v programskem jeziku Java
- **Distribucija programske in testne kode**. Testni primeri so narejeni v ločeni razredni strukturi
- **Neodvisnost testnih primerov**. JUnit nam omogoča, da testne primere izvajamo ločeno, kar pomeni, da vrstni red testov ni pomemben.
- **Poljubno združevanje testov v testne pakete**
- **Testni rezultati so jasno vidni**

4.1.2 Stroški in koristi avtomatskega testiranja modulov

Glavna korist testiranja je ugotoviti in preprečiti napake v programski kodi. Napake, ki se ugotovijo tekom razvoja, pomenijo prihranek denarja, saj je odprava napak v fazi razvoja dosti cenejša, od odprave napak po tem, ko je bil produkt že dobavljen naročniku. Poleg večjega denarnega vložka, pa napake ugotovljene v fazi, ko je produkt že v produkciji, pomenijo tudi nezadovoljstvo strank, ustavitev poslovnega procesa ter novo dobavo in namestitev programske opreme. Poleg velikih prihrankov nam avtomatsko testiranje modulov prihrani tudi kopico drugih težav [18].

Težave, ki jih srečujemo pri avtomatskem testiranju modulov, se največkrat nanašajo na težavnost vzdrževanja testih primerov. V primeru, da se aplikacija hitro spreminja, je vzdrževanje testov težavno, saj jih je potrebno popravljati in dopolnjevati z vsako spremembo, kar pa tudi zvišuje stroške avtomatizacije.

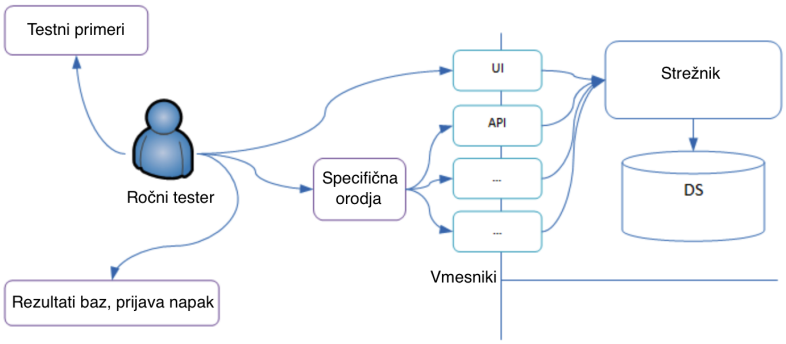
Vsemu navkljub pa je avtomatizacija testiranja tako rekoč nujna v okoljih agilnega razvoja programske opreme, saj se v procesu neprestane integracije programske kode, testi izvršijo nekajkrat dnevno, in večkrat kot se testi izvedejo, nižji so stroški avtomatizacije testiranja programske opreme.

4.2 Avtomatsko funkcionalno testiranje

Funkcijsko testiranje ali testiranje po metodi črne skrinjice je proces zagotavljanja kakovosti, s katerim preverimo, da funkcionalnosti, ki jih produkt omogoča končnemu uporabniku, delujejo natačno, zanesljivo, predvidljivo in varno. Funkcionalno testiranje lahko vključuje bodisi ročno bodisi avtomatsko testiranje. Kakorkoli, funkcionalno testiranje pomeni vrsto testov, ki posnemajo interakcijo med uporabnikom in aplikacijo, ter s tem zagotavljanja delovanje aplikacije za namen, za katerega je bila razvita.

4.2.1 Zakaj avtomatizirati funkcionalno testiranje

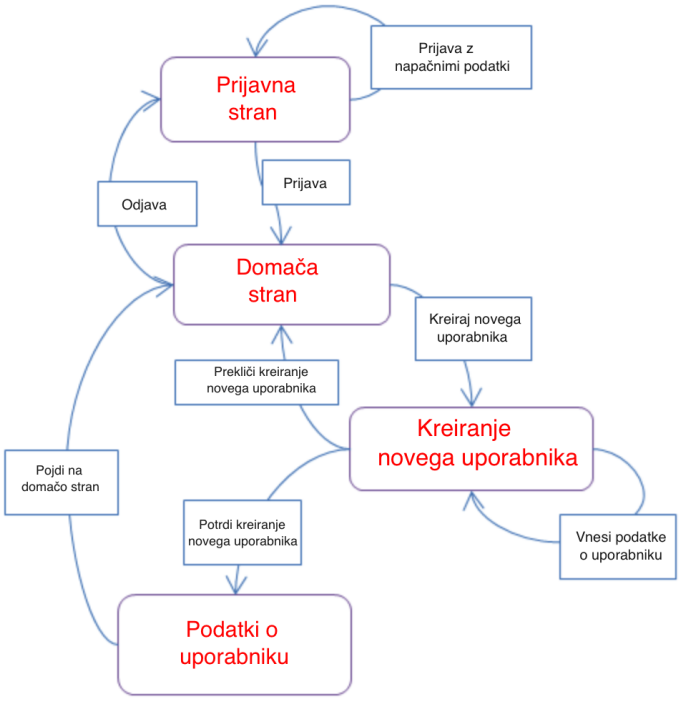
Kot je bilo omenjeno na začetku poglavja, je ročno testiranje sicer pomembno, vendar je v nekaterih primerih časovno potraten in težaven proces, kar je v nasprotju z današnjimi kratkimi časovnimi cikli razvoja programske opreme. Poleg tega z ročnim testiranjem ni mogoče temeljito preveriti delovanja aplikacije. Testiranje aplikacije, ki deluje na različnih platformah, obseg testiranja hitro povečuje. Razlog zakaj avtomatizirati testiranje, se skriva tudi v človeški nedoslednosti in napakah, ki privedejo do napak v procesu testiranja.



Slika 4.2: *Interakcija med testerjem in testnim okoljem aplikacije*

Avtomatizirano testiranje prinaša učinkovitost, ki pospešuje življenski cikel testiranja in kvaliteto razvite programske opreme. Z avtomatskimi regresijskimi testi omogočimo testerjem, da se bolj osredotočijo na manjši del testnih primerov, ki jih z avtomatskimi testi ne moremo doseči ali na določene testne primere, ki v prejšnjih verzijah niso bili preizkušeni.

Ponazorimo preprost primer prijavne strani, ki jo vsebuje skorajda že vsaka spletna aplikacija. Akcije, ki so ponazorjene s puščicami, so testni primeri, ki jih lahko avtomatiziramo.



Slika 4.3: *Primer spletne aplikacije z minimalnim naborom strani ter funkcij.*

Avtomatizacija testiranja prinese dolgoročne prednosti:

- **Ponovljivost** - avtomatsko testiranje zagotavlja dosledno sledenje strogih rokov za dobavo programske opreme, saj nam omogoča ponovno uporabo testov, kar pomeni, da ni potrebno vložiti veliko napora v ponovno testiranje. Izkušnje kažejo na to, da regresijski testi omogočajo ugotavljanje in odpravo večine napak že v zgodnji fazi razvoja programske opreme, kar omogoča gradnjo testnih paketov, ki imajo dolgoročno vrednost.

- **Predvidljivost in konsistentnost** - avtomatizirane testne primere lahko poganjamo zelo pogosto, kar je izrednega pomena, ko sistemski inženirji nameščajo dnevno izdajo (angl. build). Z regresijskimi testi lahko relativno hitro preverimo delovanje že obstoječih funkcionalnosti, ter pridobimo zgodnje informacije o delovanju novih funkcionalnosti,

kar pospešuje reševanje napak.

- **Produktivnost** - avtomatsko testiranje ustvarja izredno produktivno okolje, v katerem lahko podjetja povečajo kapacito testiranja brez dodatnih sredstev. Z avtomatizacijo, na primer, lahko preizkusimo delovanje aplikacije na različnih platformah, okoljih ter brskalnikih hkrati. To razbremeni osebje, ki se lahko osredotoča na druga vprašanja kakovosti programske opreme. Večja produktivnost prinese skrajšanje življenskega cikla testiranja, ter povečuje možnost za optimizacijo programske opreme.

4.2.2 Katere aplikacije testirati?

Tveganost vpeljave avtomatskega funkcionalnega testiranja in potrebni vložek sta veliko večja v primerjavi z avtomatizacijo modulov, zato se moramo avtomatizacije lotiti na tak način, da se tveganja minimizirajo [7].

Najprimernejše za avtomatizacijo so poslovne aplikacije, ki imajo v svoji življenski dobi več izdaj, bodisi zaradi nove, razširjene ali spremenjene funkcionalnosti. Poleg poslovnih aplikacij so primerne za avtomatizacijo tudi aplikacije, ki uporabljajo relativno stabilne podatke. Te karakteristike aplikacij omogočajo polni izkoristek ponovljivosti ter predvidenih koristi avtomatskih testov.

4.2.3 Pristopi k testiranju

Izbira pristopa je naslednja od pomembnih odločitev pred začetkom avtomatskega testiranja. Osnovni pristopi k avtomatizaciji funkcionalnih testov so *posnemi in predvajaj*, *podatkovno zasnovana arhitektura* ter *ogrodno zasnovana arhitektura*.

Posnemi in predvajaj

Pristop posnemi in predvajaj se na prvi pogled zdi precej preprost, vendar velikokrat temu ni tako. Pri tem pristopu moramo zelo paziti kaj bomo dejan-

sko testirali, saj se nam kaj hitro lahko zgodi, da bodo imele posnete skripte zelo kratek življenski cikel, zaradi težkega vzdrževanja in nestabilnosti. Ta pristop se ponavadi uporablja v zaključnih fazah projekta, ko je okolje dovolj stabilno, ter ima dovolj razvitih funkcionalnosti za smotrno izvajanje testnih skript. Primeren je predvsem za aplikacije, kjer se uporabniški vmesnik zelo malo spreminja.

Podatkovno zasnovana arhitektura

Pristop s podatkovno zasnovano arhitekturo omogoča ločevanje testne logike od testnih podatkov. Ob zagonu skripte se prebere zapis iz izhodne datoteke, kjer so shranjeni podatki in se ponovi tolikokrat, dokler niso prebrani vsi zapisi. Ta pristop zmanjšuje obseg fiksno kodirane (angl. *hard coded*) kode in omogoča hitro dodajanje novih testnih primerov, vendar so za implementacijo potrebna znanja programiranja. To se odraža tudi v vzdrževanju posameznih testnih primerov, ki morajo biti v skladu z spremembami v aplikaciji.

Ogrodno zasnovana arhitektura

Medtem ko podatkovno zasnovana arhitektura omogoča zmanjšanje obsega fiksno kodirane programske kode, pa ne obravnava neučinkovitosti kode po integraciji v skupni sistem. Ta problem rešuje ogrodno zasnovana arhitektura, katero ogrodje sestavljajo različni moduli; od preprostih, ki izvedejo točno določeno akcijo, do skript, ki izvedejo zaokroženo nalogo. Naloga bi lahko bila, npr. priprava plačilnega naloga v elektronski banki. Skripta *pripraviPlacilniNalog(a,b,c,d)*, lahko vsebuje različne ukaze, kot so: odpiranje plačilnega naloga, vnos podatkov ter avtorizacija.

4.2.4 Izbira orodja

Pomemben del odločitve za avtomatizacijo funkcionalnih testov je tudi izbira orodja. Pri orodjih za avtomatizacijo funkcionalnih testov gre ponavadi za to, da posnamemo testni primer in ga kasneje poljubno mnogokrat poženemo.

Omogočajo distribuirano izvajanje, kot tudi poganjanje celotnih naborov testnih primerov. Orodja ponavadi vsebujejo tri osnovne funkcije:

- **Priprava testov** - test je pripravljen, ko ga posnamemo skozi interakcijo z testirano aplikacijo. Posneti modul generira kodo v ozadju, katero lahko kasneje poljubno spreminjamo v integriranem orodju za urejanje skript. V primeru, da imamo podatkovno vodene teste, nam orodja za avtomatizacijo omogočajo povezovanje pripravljenih testov z notranjim ali zunanjim izvorom podatkov.
- **Izvedba testov** - testi se lahko poženejo avtomatsko, lahko pa jih poženemo tudi ročno.
- **Poročanje rezultatov** - ko se testiranje zaključi, se nam generira poročilo izvedenih testov, ki nam omogoča hiter vpogled v delovanje testirane aplikacije.

Pomembne značilnosti orodij za avtomatizacijo funkcionalnega testiranja so predvsem *dobro razpoznavanje objektov*, *sinhronizacija testiranja*, *preverjanje rezultatov*, *programski jezik orodja* ter *obravnava napak*.

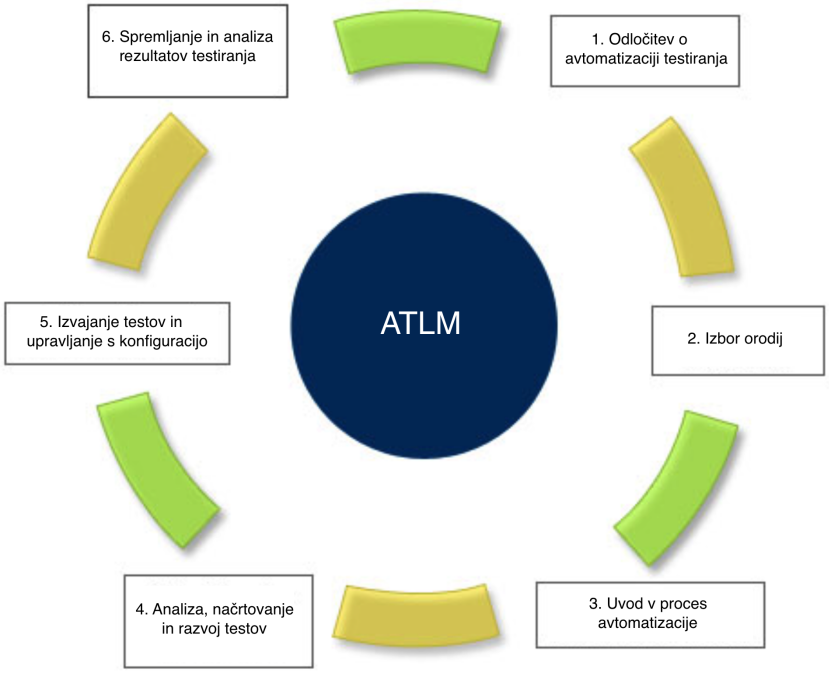
4.3 Metodologija ATLM

Metodologija ATLM (angl. *Automation Testing Life-cycle Methodology*) predstavlja strukturiran pristop k implementaciji avtomatizacije testiranja. Pristop vključuje večstopenjski proces, podporo podrobnim ter medsebojno povezanim dejavnostim, ki so potrebne za uvedbo testnega orodja za avtomatizacijo, oblikovanje testnih primerov, razvoj in izvajanje testnih primerov, ter razvoj in upravljanje testnih podatkov in testnega okolja [10].

Metodologijo ATLM sestavlja šest osnovnih faz:

- Odločitev o avtomatizaciji testiranja
- Izbor orodij

- Uvod v proces avtomatizacije
- Analiza, načrtovanje in razvoj testov
- Izvajanje testov in upravljanje s konfiguracijo
- Spremljanje in analiza rezultatov testiranja



Slika 4.4: *Šest osnovnih faz metodologije ATLM.*

4.3.1 Odločitev o avtomatizaciji testiranja

Odločitev o avtomatizaciji testiranja predstavlja prvo fazo strukturirane metodologije ATLM, ki zajema celoten proces odločitve za avtomatsko testiranje. V tej fazi mora testna ekipa definirati svoja pričakovanja glede avtomatskega testiranja in predstaviti koristi avtomatizacije ob njeni pravilni implementaciji.

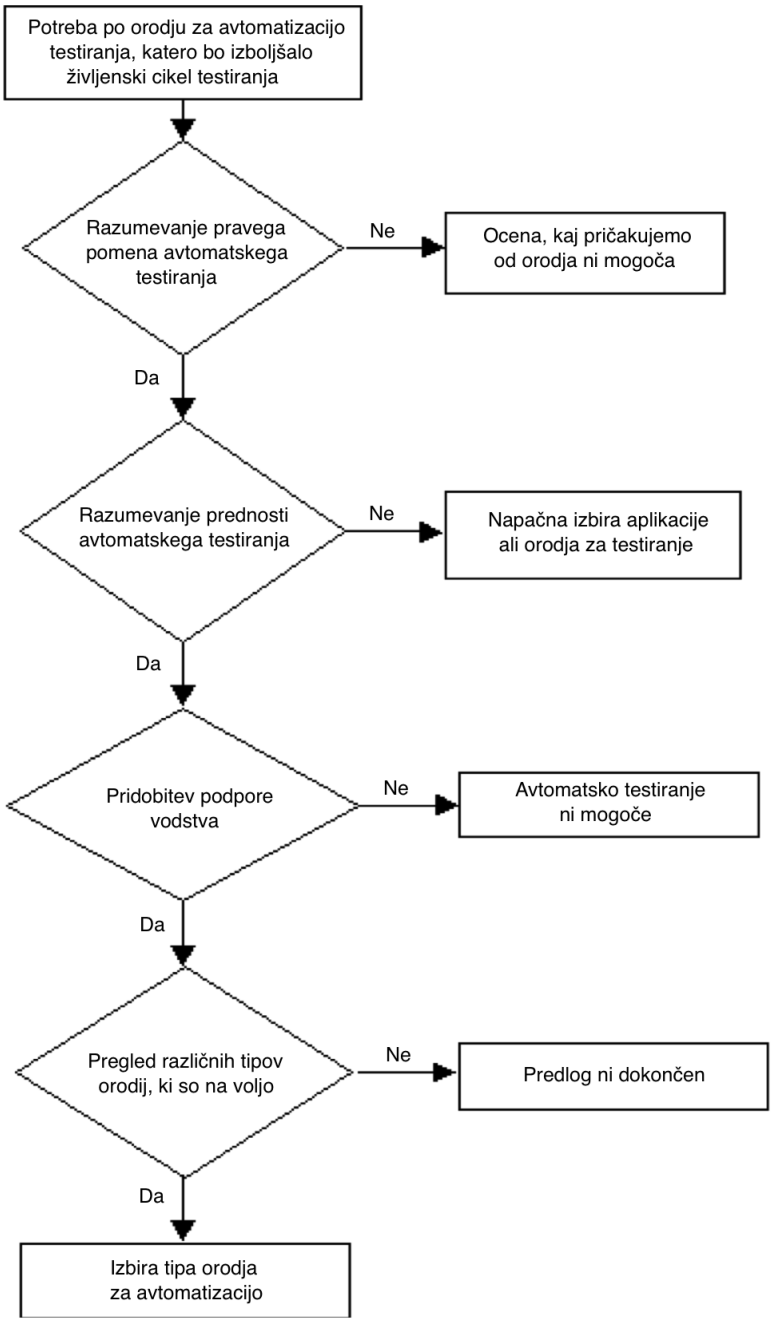
Po tem, ko je sklenjen dogovor, da avtomatizacija testiranja prinaša dodano vrednost, se je potrebno zavedati, da se naložba v nekaterih primerih

ne povrne takoj. Odgovorne osebe se morajo zavedati, da avtomatsko testiranje zahteva nekaj vloženega časa in energije, da se doseže dolgoročna donosnost naložbe (angl. *Return Of Investment*). Vse prepogosto se zgodi, da so predstave o avtomatskem testiranju napačne. Nekatere izmed napačnih predpostavk ter mitov avtomatskega testiranja so:

- **Avtomatska izdelava testnega načrta** - trenutno ni na voljo nobenega komercialnega orodja, ki bi omogočalo samodejno izdelavo celovitega testnega načrta, kateri bi hkrati podpiral izdelavo testov in izvajanje. Zavedati se namreč moramo, da avtomatsko testiranje ne nadomesti ročnega testiranja v celoti, to je še vedno potrebno za testiranje produkta. Testno orodje lahko gledamo kot del ekipe, ki je potrebna, da zagotovi dober in kvaliteten produkt.

- **Orodje pokriva vse platforme** - veliko orodij podpira širok spekter operacijskih sistemov in platform, vendar pa trenutno še ne obstaja eno samo orodje, ki bi pokrivalo celoten nabor platform.

- **Avtomatizacija vseh testov** - avtomatizacija vseh testnih primerov v praktičnem okolju ni mogoča in tudi ne stroškovno upravičena.



Slika 4.5: *Potek odločitve o avtomatizaciji testiranja.*

4.3.2 Izbira orodij

Faza izbire orodij vodi testnega inženirja skozi celoten ocenitveni postopek testnega orodja in izbranega pristopa k testiranju. Potem, ko slednji ugotovi, da orodje zadostuje večini testnih zahtev podjetja, mora preučiti še sistemsko okolje in druge zahteve, ter narediti seznam kriterijev za izbiro orodja. Ponavadi so preizkusna obdobja zelo kratka, zato je potrebno v kratkem času narediti nekaj delujočih testov, ter preizkusiti delovanje na infrastrukturi, ki nam je na voljo.

4.3.3 Uvod v proces avtomatizacije

Ta faza opisuje korake, ki so potrebni za uspešno vpeljavo avtomatskega testiranja v nov ali obstoječ projekt.

Analiza procesa testiranja
Analiza procesa testiranja zagotavlja, da se držimo procesa testiranja, in da se proces po potrebi dopolnjuje in spreminja, kar omogoča uspešno vpeljavo avtomatskih testov. Testni inženirji opredelijo in zbirajo testne metrike v želji po izboljšanju procesa avtomatizacije. Testni proces mora biti dobro dokumentiran, da ga lahko predstavimo vsem vpletenim. V primeru, da testni proces ni dobro dokumentiran, ni ponovljiv niti razumljiv udeležencem, kar posledično pripelje do tega, da se proces ne bo izvajal. Nedokumentiranega procesa prav tako ne moremo avtomatizirati in izboljševati.

Obravnava testnih orodij
V fazi izbire orodij, testna ekipa definira, katera orodja splošno podpirajo proces testiranja v organizaciji. V tej fazi pa je potrebno preveriti ali so orodja resnično primerna za ciljni projekt. Pri tem je potrebno upoštevati podane zahteve projekta, testna okolja, ter število testerjev, ki bo delalo na projektu. Odgovornim je potrebno predstaviti časovnico vzpostavitve testnega orodja in preizkusa združljivosti orodja z obstoječo infrastrukturo.

4.3.4 Analiza, načrtovanje in razvoj testov

Planiranje testiranja predvideva natančen pregled celotnega procesa in aktivnosti, potrebnih za izvajanje testiranja in verifikacije. Predvideva tudi, da bodo procesi, strojna in programska oprema ter omrežje, ki so potrebni za podporo testnega okolja vključno s podatki, organizirani in uporabljeni na učinkovit način.

Planiranje testiranja vsebuje vse rezultate predhodnih faz ATLM metodologije. V fazi planiranja se definirajo vse vloge in odgovornosti, tveganje testiranja ter sprejemljiv nivo delovanja produkta.

Analiza in načrtovanje testov

Podobno kot razvoj programske opreme, tudi avtomatizacija testiranja zahteva celovit pristop, in sicer z jasno definiranimi zahtevami, analizo, načrtovanjem in kodiranjem. Po definiranju zahtev se prične načrtovanje testnih primerov, pri čemer moramo definirati tudi ali gre pri določenem testnem primeru za avtomatski ali ročni test. Testna skupina na ta način dobi vpogled v to, koliko testov bo potrebno opraviti ročno in koliko testnih primerov bo izvedenih avtomatsko. Pri načrtovanju avtomatskih testov je potrebno dokončno definirati in dokumentirati način kodiranja ter pristop k izdelavi programske kode testov, ki bo zagotavljal ponovljivost testov, ter čim bolj enostavno vzdrževanje.

Sledi definicija testnih primerov, ki zajema podatke o avtorju, predpogojih, vhodih, izhodih, potrebnih akcijah in dosegu testiranja. Podrobnost opisa testnega primera je odvisna od njegove kompleksnosti. Bolj zapletene primere je potrebno označiti in jih podrobneje preučiti. Testne primere je na koncu smiselno združiti v logične skupine in jih povezati z viri testnih podatkov.

Razvoj testnih primerov

Po analizi in načrtovanju testnih primerov je testna skupina pripravljena na njihov razvoj. Pri razvoju se je potrebno držati prej definiranih pravil glede

načina kodiranja in pristopa k izdelavi testov. Pri kodiranju moramo paziti na to, da bodo testni primeri primerni za regresijsko testiranje, in da jih bo kasneje relativno lahko vzdrževati. Potrebno se je držati tudi časovnega okvirja in se mu ustrezno prilagajati. Za vsako ugotovljeno težavo je potrebno dokumentirati, koliko časa je potekalo njeno odpravljanje in koliko časa ponovna izvedba testov.

4.3.5 Izvajanje testov in upravljanje s konfiguracijo

Po končanem razvoju testnih primerov so slednji pripravljeni za izvedbo na testnem okolju aplikacije. Z izvajanjem testnih primerovse hkrati zbirajo in analizirajo tudi rezultati. V tej fazi je zajeto testiranje na vseh ravneh, od testiranja modulov, integracijskih testov, sistemskih testov do sprejemnih testov. Plan testiranja testov je v zaporedju, ki je preddefiniran in potreben, da se testi izvedejo na celotnem sistemu, saj je s tem zagovotvljena kvaliteta produkta na vseh ravneh. Testna skupina je odgovorna za odkrivanje in poglobljeno testiranje komponent sistema, pri katerih največkrat pride do napak. Po končanem testiranju se beležijo napake ter dopolnjuje dokumentacija.

Vodja testiranja je odgovorna, da testiranje poteka v skladu z urnikom in ustrezno razporedi testerje na posamezne sklope testiranja ter jih prerazporeja, če se v fazi testiranja pojavijo težave. Za prikaz ključnih indikatorjev testiranja (količina testnih primerov, napredek ter uspešnost testiranja,...), skrbijo testne metrike.

Nekatere izmed standardnih testnih metrik pri testiranju programske opreme so:

- Število odkritih napak v okviru dnevne izdaje
- Število odkritih napak v okviru posameznega modula
- Čas do odkritja napak
- Prioriteta reševanja odkritih napak

- Čas za zagotavljanje testov
- Število napak v okviru enega testiranja

4.3.6 Spremljanje in analiza rezultatov testiranja

Zadnja faza vpeljave avtomatskega testiranja je spremljanje in analiza rezultatov testiranja, ki je potrebna za nadaljevanje z izboljšavami na projektu ter drugimi aktivnosti. Analiza rezultatov zmogljivostnega testiranja (angl. *performance testing* zagotavlja odkritje delov sistema, kjer lahko na naslednjem projektu še izboljšamo rezultate zmogljivosti. Pri analizi rezultatov se ocenjuje, ali delovanje aplikacije ustreza predpisanim kriterijem, ter ali je primerna za namestitev v produkcijsko okolje.

Testna skupina mora sprejeti iterativni proces učenja kot del svoje kulture, saj je potrebno dokumentirati tako pozitivne kot negativne izkušnje ter izvedene izboljšave. Poleg dokumentacije je potreben tudi izračun povračila naložbe v avtomatizacijo, za kar moramo izbrati ustrezne podatke že tekom procesa testiranja.

Poglavje 5

Implementacija v podjetju

Podjetje Halcom d.d., je IT podjetje, ki že več kot dvajsetletje ponuja rešitve za finančne ustanove. V tem poglavju bom opisal vpeljavo funkcionalnega avtomatskega testiranja, v proces produktno-projektnega načina dela, v podjetju Halcom.

5.1 Avtomatizacija testiranja v produktnem načinu dela

V svetu elektronskega bančništva, se velikokrat srečujemo z venomer enakimi funkcionalnostmi na različnih bankah. Zato, se je podjetje Halcom odločilo, da želi imeti produkt, ki bo zajemal sklop vseh funkcionalnosti, ki jih banke v svojih elektronskih rešitvah želijo, jih dopolnjeval in izpopolnjeval v skladu s smernicami, ki narekujejo kakšno elektronsko bančništvo sledi v prihodnosti, ter na ta način bankam zagotavljal poljuben obseg implementacije funkcionalnosti, ki jih banka želi.

Razvoj programske opreme v podjetju poteka po metodologiji agilnega razvoja *Scrum*, katerega pomemben del je tudi avtomatizacija testiranja. Po analizi koristi, ki bi jih prineslo avtomatsko testiranje, analizi okolja na katerem se bo testiranje izvajalo, ter analizi orodij, ki bi nam omogočala ustrezno implementacijo testov, smo se za slednje tudi odločili.

5.1.1 Izbira orodja

Pri izbiri orodja smo se osredotočali na podlagi naslednjih metrik:

- robustno testiranje aplikacije preko grafičnega uporabniškega vmesnika
- enostavna vključitev v proces razvoja
- odlično razpoznavanje objektov
- podatkovno vodeno testiranje
- regresijski testi
- širok spekter testiranja na različnih platformah in spletnih brskalnikih
- hiter in učinkovit pregled rezultatov
- redne posodobitve orodja
- podpora strankam

Na podlagi naštetih metrik, smo se odločili za orodje **Ranorex**. Ranorex namreč omogoča odlično razpoznavanje objektov, ki so neodvisni od tega v katerem spletnem brskalniku jih posnamemo, implementacijo robustnih in zanesljivih testov, ter enostavno integracijo v obstoječe razvojno okolje. Prav tako podpira, skorajda celoten spekter spletnih tehnologij, ki krojijo današnji svet razvoja spletnih aplikacij.

Ranorex omogoča preprosto snemanje modulov preko grafičnega uporabniškega vmesnika, v ozadju pa sam generira skriptno kodo v programskem jeziku C#. Za zahtevnejše uporabnike je v orodju implementiran tudi urejevalnik kode, preko katerega lahko razvijalec sam napiše testni primer, ki ga želi izvesti, vendar za to potrebuje nekaj programerskega znanja [3].

Slika 5.1: *Spletni brskalniki in tehnologije, ki jih podpira orodje Ranorex.*

Hierarhično sestavljanje testnega paketa, v katerega so vključeni testni razredi, ki so sestavljeni ponavadi iz več testnih modulov, je naslednja dobra lastnost tega orodja. Na nivoju testnega paketa, lahko definiramo statične, kot tudi dinamične globalne spremenljivke, ki so dosegljive znotraj vseh razredov in modulov, ki so zajeti v testnem paketu. Za definiranje dinamičnih spremenljivk moramo v datoteki *Program.cs*, ki je del vsakega testnega paketa, sprogramirati funkcijo, ki nam bo ob vsakem zagonu testnega paketa, generirala ustrezne spremenljivke. Poleg globalnih spremenljivk, lahko določimo tudi lokalne spremenljivke, katere se dodajajo na različnih nivojih testnih razredov.

Poglejmo preprost primer datoteke Program.cs, ki nam vsakič ob zagonu testnega paketa definira globalni spremenljivki "today"in "tomorrow", ki nosita dejanski vrednosti današnjega ter jutrišnega datuma.

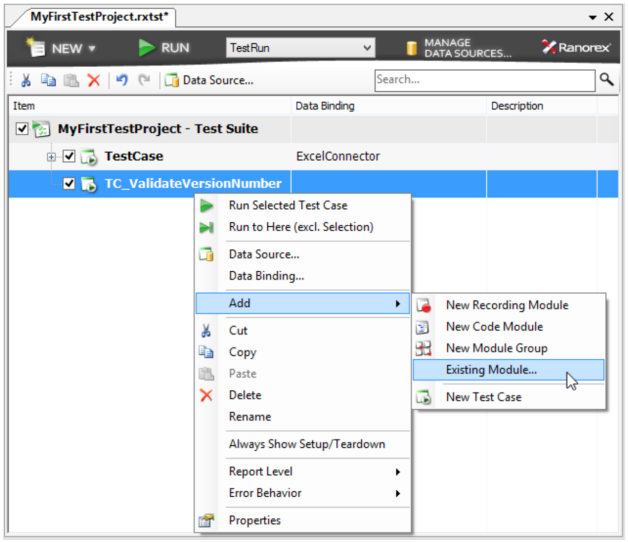
```
using System;
using System.Threading;
using System.Drawing;
```

```
using System.Collections.Generic;
using System.Text.RegularExpressions;
using WinForms = System.Windows.Forms;
using Ranorex;
using Ranorex.Core;
using Ranorex.Core.Reporting;
using Ranorex.Core.Testing;

namespace Name_Of_Your_Solution{
    class Program
    {
    [STAThread]
    public static int Main(string[] args){
        Keyboard.AbortKey = System.Windows.Forms.Keys.Pause;
        int error = 0;
        String commandLineArgument;
        if (Environment.CommandLine.Length == 0){
            commandLineArgument = "date.exe /pa:today=" +
                System.DateTime.Today.ToString("yyyy-MM-dd") +
                " /pa:tomorrow=" +
                    System.DateTime.Now.AddDays(1).ToString("yyyy-MM-dd");
        }
        else{
                commandLineArgument = "date.exe /pa:today=" +
                    System.DateTime.Today.ToString("yyyy-MM-dd") +
            " /pa:tomorrow=" +
                System.DateTime.Now.AddDays(1).ToString("yyyy-MM-dd");
        }
    try
    {
        error = TestSuiteRunner.Run(typeof(Program),
            commandLineArgument);
    }
```

```
catch (Exception e)
{
    Report.Error("Unexpected exception occurred: " +
        e.ToString());
    error = -1;
}
return error;}}}
```

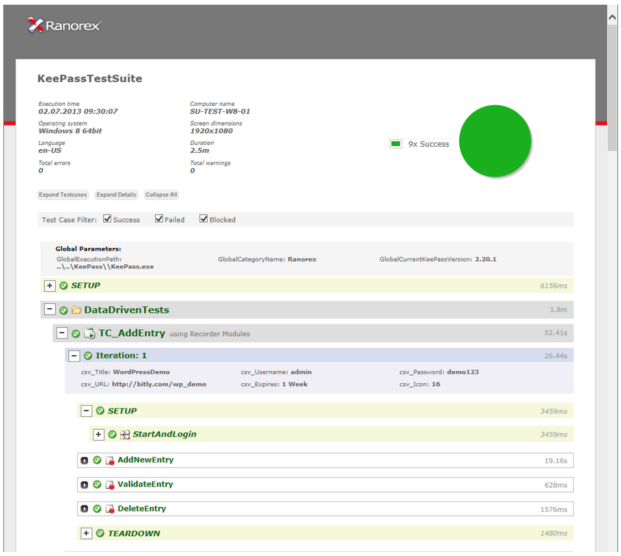
Ob kreiranju novega testnega paketa, nam ranorex omogoča uvoz že obstoječega repozitorija in pripadajočih modulov, kar v produktno-projektnem načinu dela, s predpostavko, da imamo funckionalno testiranje produkta že avtomatizirano, zelo olajša delo razvijalcu testnih primerov. Poljuben del standardne rešitve, ki bo implementirana pri stranki, namreč lahko preprosto vzamemo iz že obstoječe rešitve, in po potrebi razvijemo in dopolnimo samo projektno specifične testne primere.



Slika 5.2: *Dodajanje obstoječega modula v testni razred, znotraj testnega paketa.*

Med pomembne korake pri avtomatskem testiranju sodi tudi analiza rezultatov. Orodje Ranorex po koncu vsakega izvajanja testnih primerov naredi poročilo, ki vsebuje relativno natančen opis izvedenih akcij v aplikaciji,

tekom avtomatskega testiranja.



Slika 5.3: *Generiranje rezultatov po končanem izvajanjem testov v orodju Ranorex.*

Orodje Ranorex poleg omenjenih funkcionalnosti omogoča še vrsto drugih, ki pa v pričujočem delu ne nosijo večjega pomena, zato nadaljnjo raziskavo orodja prepuščam bralcu.

5.1.2 Proces avtomatizacije

Ob vprašanju kdaj začeti z avtomatizacijo se pogosto porodi kopica drugih vprašanj. Ena izmed dilem, ki jo srečujemo pri avtomatskem funkcionalnem testiranju preko grafičnega uporabniškega vmesnika je ta, da pri tem načinu avtomatizacije potrebujemo stabilno okolje, ki ga v veliki večini primerov v verifikacijskem okolju še nimamo, zato smo pogosto prisiljeni začeti avtomatizacijo testiranja v zaključnih fazah razvoja projekta, ki ponavadi poteka v validacijskem okolju.

Na ta problem smo v podjetju Halcom odgovorili tako, da v verifikacijsko okolje prihajajo razviti manjši sklopi funkcionalnosti, ki nam omogočajo celotno avtomatizacijo testiranja razvitega sklopa. Ko pride naslednji sklop,

preverimo delovanje obstoječega sklopa z že implementiranimi avtomatskimi testi, avtomatizacijo testov novega sklopa pa razvijamo vzporedno, ter na koncu avtomatizirana sklopa združimo. Ta proces se ponavlja, dokler ne razvijemo vseh potrebnih funkcionalnosti, nato pa aplikacijo namestimo v validacijsko okolje, kjer lahko zopet izvedemo vse avtomatske teste, ki smo jih razvili na verifikacijskem okolju, ter s tem preverimo funkcionalnost delovanja aplikacije. Poleg tega na ta način razvojna skupina dobi zelo ažurno informacijo o delovanju aplikacije.

Seveda ne smemo pozabiti, da se poleg avtomatskega testiranja aplikacij izvaja tudi ročno testiranje, ker kot smo že omenili v predhodnjih poglavjih, avtomatsko testiranje ne more v celoti nadomestiti ročnega testiranja. Ročno se testira predvsem tiste dele aplikacije, ki jih z avtomatskimi testi ne dosežemo. Ročno testiranje izvajajo izkušeni testerji, ki natančno poznajo funkcijske specifikacije ter delovanje produkta, avtomatski testi pa jim omogočajo, da se še bolj osredotočijo na kvalitetno in zanesljivo delovanje aplikacije, več časa pa ostane tudi za analizo rezultatov, ter dokumentiranje le teh.

5.1.3 Vzdrževanje

Za vzdrževanje testnih primerov skrbimo z sistemom za verzioniranje programske kode Git. V Git repozitorij se namreč odlagajo vse spremembe, ki so bile narejene v kodi testnih primerov, hkrati pa nam omogoča hitro odpravo napak ter dodajanje ali ažuriranje testnih primerov. Poleg verzioniranja kode, pa je potrebno pred začetkom avtomatskega testiranja imeti dobro pripravljeno testno okolje, za kar ponavadi poskrbijo sistemski inženirji.

5.1.4 Koristi avtomatizacije

O razlogih in koristih za avtomatizacijo testiranja programske opreme je bilo že veliko napisanega v prejšnjih poglavjih in tudi v primeru podjetja Halcom ni temu nič drugače. Koristi ki jih zaznavamo tekom procesa razvoja se

kažejo predvsem v naslednjih metrikah:

- **Prihranek časa** - je ena glavnih prednosti, ki se še posebno kaže pri regresijskem testiranju.
- **Hitrost izvajanja testov** - teste izvaja izbrano orodje za avtomatizacijo, ki neprimerno hitreje izvede testne primere, kot če bi bilo potrebno ročno preverjanje le teh.
- **Vzdrževanje** - v produktno-projektnem načinu dela, se avtomatski testi izvajajo zelo pogosto, zato smo primorani ustrezno vzdrževati testne primere, da vedno ustrezajo zadnjim popravkom v programski kodi, ter dodajati nove specifične module.
- **Ponovljivost testov** - teste lahko uporabimo na različnih projektih, ki imajo določen nabor skupnih funkcionalnosti, lahko pa jih uporabimo tudi na različnih verzijah programske opreme.
- **Zniževanje stroškov** - dandanes še kako velja rek *"čas je denar"*. Avtomatizacija testiranja prihrani veliko časa, poleg tega omogoča, da se testerij osredotočijo na kritične teste in napake v aplikaciji, kar je pomembno, saj je odkrivanje in reševanje napak, ko je aplikacija že v produkciji, težavno in eksponentno povečuje stroške projekta.

Poglavje 6

Zaključek

Pojav agilnih metodologij je nedvomno prinesel spremembe v razvoj programske opreme. Agilne metodologije zagovarjajo predvsem hitre in stalne izdaje programske opreme, kar je posledica tega, da se v današnjem hitro razvijajočem se svetu, uporabniške zahteve zelo hitro spreminjajo. Zaradi slednjega je avtomatizacija testiranja programske opreme praktično obvezna. Metodologija ekstremnega programiranja ga celo postavlja med svoje ključne procese.

Avtomatizacija testiranja ima v splošnem kar nekaj pasti, zato je potrebno biti previden in pozoren pri analiziranju vseh potrebnih dejavnikov, ki pogojujejo uspešno vpeljavo avtomatizacije testiranja programske opreme. Če je analiza dejavnikov dobra, se nam koristi avtomatizacije testiranja kmalu nasmehnejo. V diplomskem delu je predstavljena metodologija ATLM, ki nam pri o odločitvi za avtomatsko testiranje lahko zelo pomaga.

Cilj diplomske naloge je bil vpeljava avtomatizacije testiranja v produktno-projektni način dela v podjetju Halcom. V podjetju smo naredili analizo orodij in se na koncu odločili za orodje Ranorex, kar se je izkazalo za dobro izbiro, saj nam zagotavlja vse, kar potrebujemo za uspešno implementacijo funkcionalnih avtomatskih testov. Testiranje preko grafičnega uporabniškega vmesnika v podjetju začnemo že v verifikacijskem okolju in s tem zagotovimo, da v validacijsko okolje pride stabilnejša verzija. Testi se izvajajo večkrat

tedensko, s čimer smo dosegli hitro odzivnost testne ekipe, pospešili odpravljanje napak in boljšo kvaliteto programske opreme.

Literatura

[1] Gnu general public licence. https://www.gnu.org/copyleft/gpl.html.

[2] Junit. http://www.tutorialspoint.com/junit/junit_overview.htm. Dostopno: 2014-08-25.

[3] Ranorex. http://www.ranorex.com. Dostopno: 2014-08-26.

[4] Test automation and software development. http://www.tcs.com/SiteCollectionDocuments/White%20Papers/Test%20Automation%20and%20Software%20Development.pdf. : 2014-08-23.

[5] Test life cycle / software testing models(manual testing). http://www.siliconindia.com/online-courses/tutorials/Test-Life-Cycle--Software-Testing-modelsmanual-testing-id-44.html. Dostopno: 2014-08-25.

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[7] Peter Čebokli. *Avtomatizacija testiranja kot ključ agilnosti razvoja programske opreme*. PhD thesis, Univerza v Ljubljani, 2006.

[8] Andraž Cej. *Agilni razvoj programske opreme po metodologiji Scrum*. PhD thesis, Univerza v Ljubljani, 2010.

[9] Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perumandla. A complete automation of unit testing for java programs. 2005.

[10] E. Dustin. The automated testing life-cycle methodology (atlm). 2000.

[11] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[12] M Kezunovic. Future trends in protective relaying, substation automation, testing and related standardization. In *Transmission and Distribution Conference and Exhibition 2002: Asia Pacific. IEEE/PES*, volume 1, pages 598–602. IEEE, 2002.

[13] Johannes Link. *Unit testing in Java: how tests drive the code*. Morgan Kaufmann, 2003.

[14] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[15] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[16] Sridhar Nerur, RadhaKanta Mahapatra, and George Mangalaraj. Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5):72–78, 2005.

[17] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[18] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91. ACM, 2006.

[19] James Shore et al. *The art of agile development*. "O'Reilly Media, Inc.", 2007.

[20] Franc Solina. Projektno vodenje razvoja programske opreme. 1997.

[21] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[22] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[23] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of gui testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14. ACM, 2006.

[24] Kevin Vlaanderen, Slinger Jansen, Sjaak Brinkkemper, and Erik Jaspers. The agile requirements refinery: Applying scrum principles to software product management. *Information and Software Technology*, 53(1):58–70, 2011.

[25] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932. ACM, 2006.

[26] Xun Yuan, Myra B Cohen, and Atif M Memon. Gui interaction testing: Incorporating event context. *Software Engineering, IEEE Transactions on*, 37(4):559–574, 2011.